Deriving Modernity Signatures for php Systems with Static Analysis

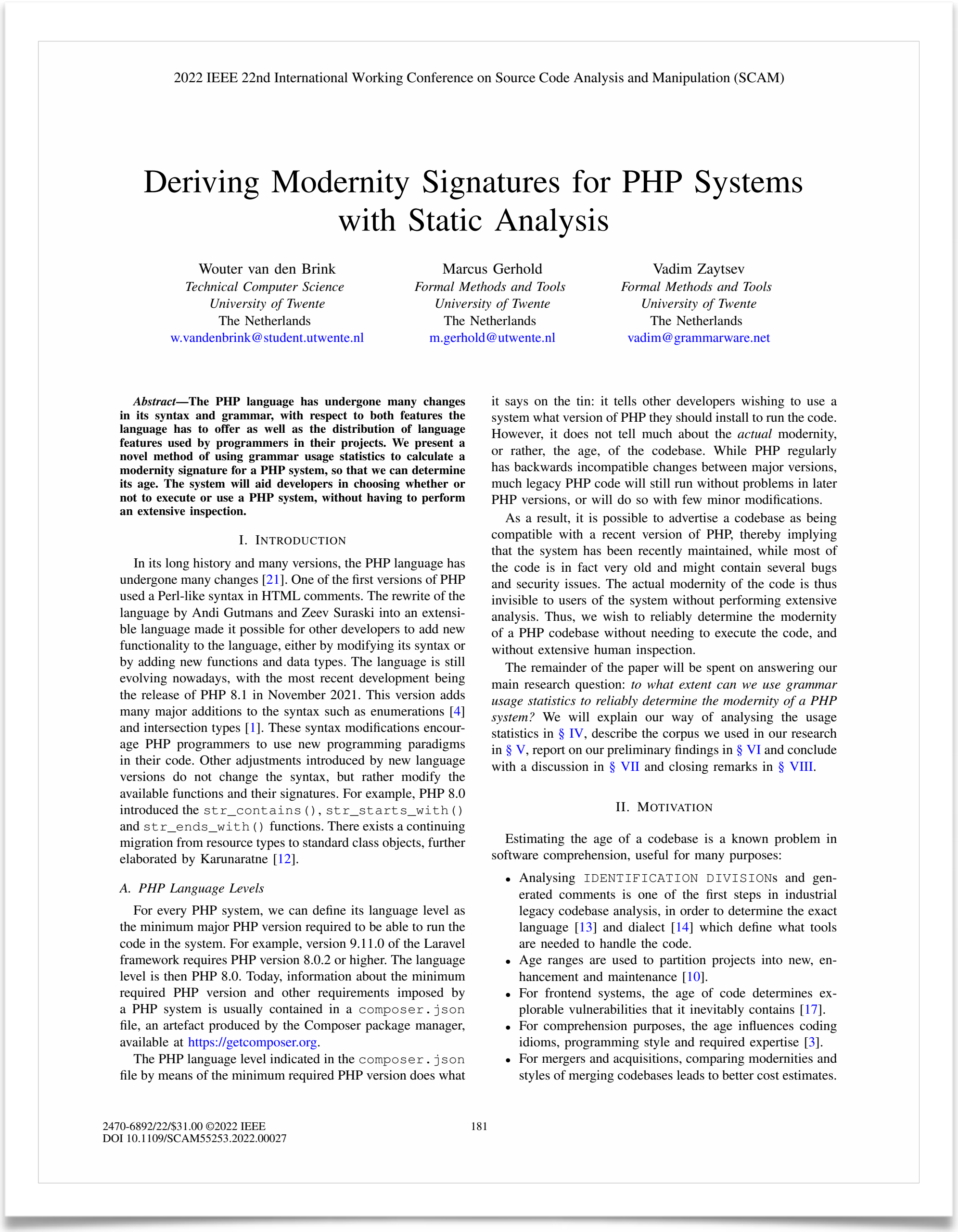
SCAM NIER, 4 October 2022

Dr. Vadim Zaytsev aka @grammarware

UNIVERSITY OF TWENTE.

What's in the Paper?

- 960900a181/960900a181.pdf
- https://doi.org/10.1109/SCAM55253.2022.00027
- pp.181–185



2022 IEEE 22nd International Working Conference on Source Code Analysis and Manipulation (SCAM)

Deriving Modernity Signatures for PHP Systems with Static Analysis

Wouter van den Brink
Technical Computer Science
University of Twente
The Netherlands
w.vandenbrink@student.utwente.nl

Marcus Gerhold
Formal Methods and Tools
University of Twente
The Netherlands
m.gerhold@utwente.nl

Vadim Zaytsev
Formal Methods and Tools
University of Twente
The Netherlands
vadim@grammarware.net

***Abstract*—The PHP language has undergone many changes in its syntax and grammar, with respect to both features the language has to offer as well as the distribution of language features used by programmers in their projects. We present a novel method of using grammar usage statistics to calculate a modernity signature for a PHP system, so that we can determine its age. The system will aid developers in choosing whether or not to execute or use a PHP system, without having to perform an extensive inspection.**

I. Introduction

In its long history and many versions, the PHP language has undergone many changes [21]. One of the first versions of PHP used a Perl-like syntax in HTML comments. The rewrite of the language by Andi Gutmans and Zeev Suraski into an extensible language made it possible for other developers to add new functionality to the language, either by modifying its syntax or by adding new functions and data types. The language is still evolving nowadays, with the most recent development being the release of PHP 8.1 in November 2021. This version adds many major additions to the syntax such as enumerations [4] and intersection types [1]. These syntax modifications encourage PHP programmers to use new programming paradigms in their code. Other adjustments introduced by new language versions do not change the syntax, but rather modify the available functions and their signatures. For example, PHP 8.0 introduced the `str_contains()`, `str_starts_with()` and `str_ends_with()` functions. There exists a continuing migration from resource types to standard class objects, further elaborated by Karunaratne [12].

A. PHP Language Levels

For every PHP system, we can define its language level as the minimum major PHP version required to be able to run the code in the system. For example, version 9.11.0 of the Laravel framework requires PHP version 8.0.2 or higher. The language level is then PHP 8.0. Today, information about the minimum required PHP version and other requirements imposed by a PHP system is usually contained in a `composer.json` file, an artefact produced by the Composer package manager, available at https://getcomposer.org.

The PHP language level indicated in the `composer.json` file by means of the minimum required PHP version does what it says on the tin: it tells other developers wishing to use a system what version of PHP they should install to run the code. However, it does not tell much about the *actual* modernity, or rather, the age, of the codebase. While PHP regularly has backwards incompatible changes between major versions, much legacy PHP code will still run without problems in later PHP versions, or will do so with few minor modifications.

As a result, it is possible to advertise a codebase as being compatible with a recent version of PHP, thereby implying that the system has been recently maintained, while most of the code is in fact very old and might contain several bugs and security issues. The actual modernity of the code is thus invisible to users of the system without performing extensive analysis. Thus, we wish to reliably determine the modernity of a PHP codebase without needing to execute the code, and without extensive human inspection.

The remainder of the paper will be spent on answering our main research question: *to what extent can we use grammar usage statistics to reliably determine the modernity of a PHP system?* We will explain our way of analysing the usage statistics in § IV, describe the corpus we used in our research in § V, report on our preliminary findings in § VI and conclude with a discussion in § VII and closing remarks in § VIII.

II. Motivation

Estimating the age of a codebase is a known problem in software comprehension, useful for many purposes:

- Analysing `IDENTIFICATION DIVISION`s and generated comments is one of the first steps in industrial legacy codebase analysis, in order to determine the exact language [13] and dialect [14] which define what tools are needed to handle the code.
- Age ranges are used to partition projects into new, enhancement and maintenance [10].
- For frontend systems, the age of code determines explorable vulnerabilities that it inevitably contains [17].
- For comprehension purposes, the age influences coding idioms, programming style and required expertise [3].
- For mergers and acquisitions, comparing modernities and styles of merging codebases leads to better cost estimates.

2470-6892/22/$31.00 ©2022 IEEE
DOI 10.1109/SCAM55253.2022.00027

Let’s Focus on the Context…

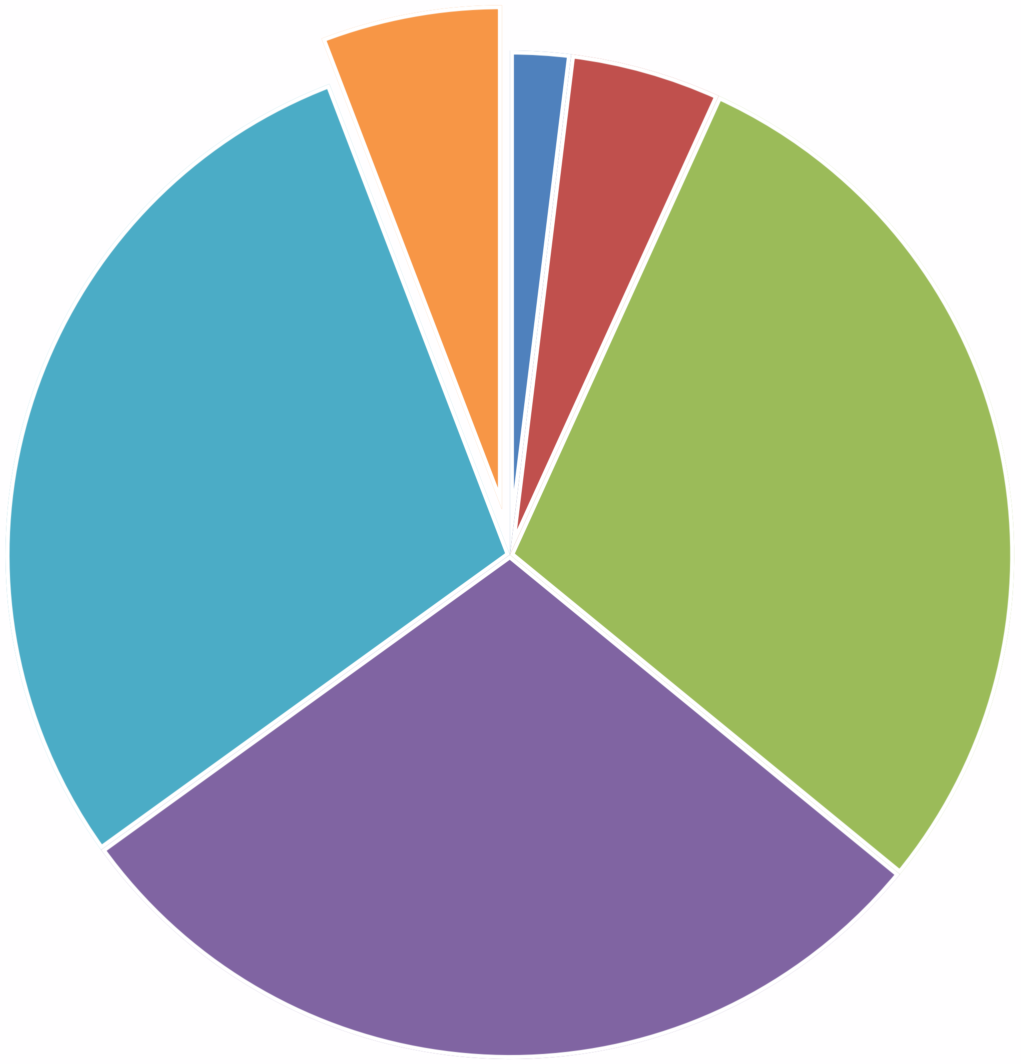
- What is the problem?

Portfolio Analysis

- be me (3+ years ago)
- analyst/developer
- new potential customer
- PoC: 10000s of files
- COBOL? COPY?
- PL/I?
- HLASM?
- REXX? CLIST? RPG?

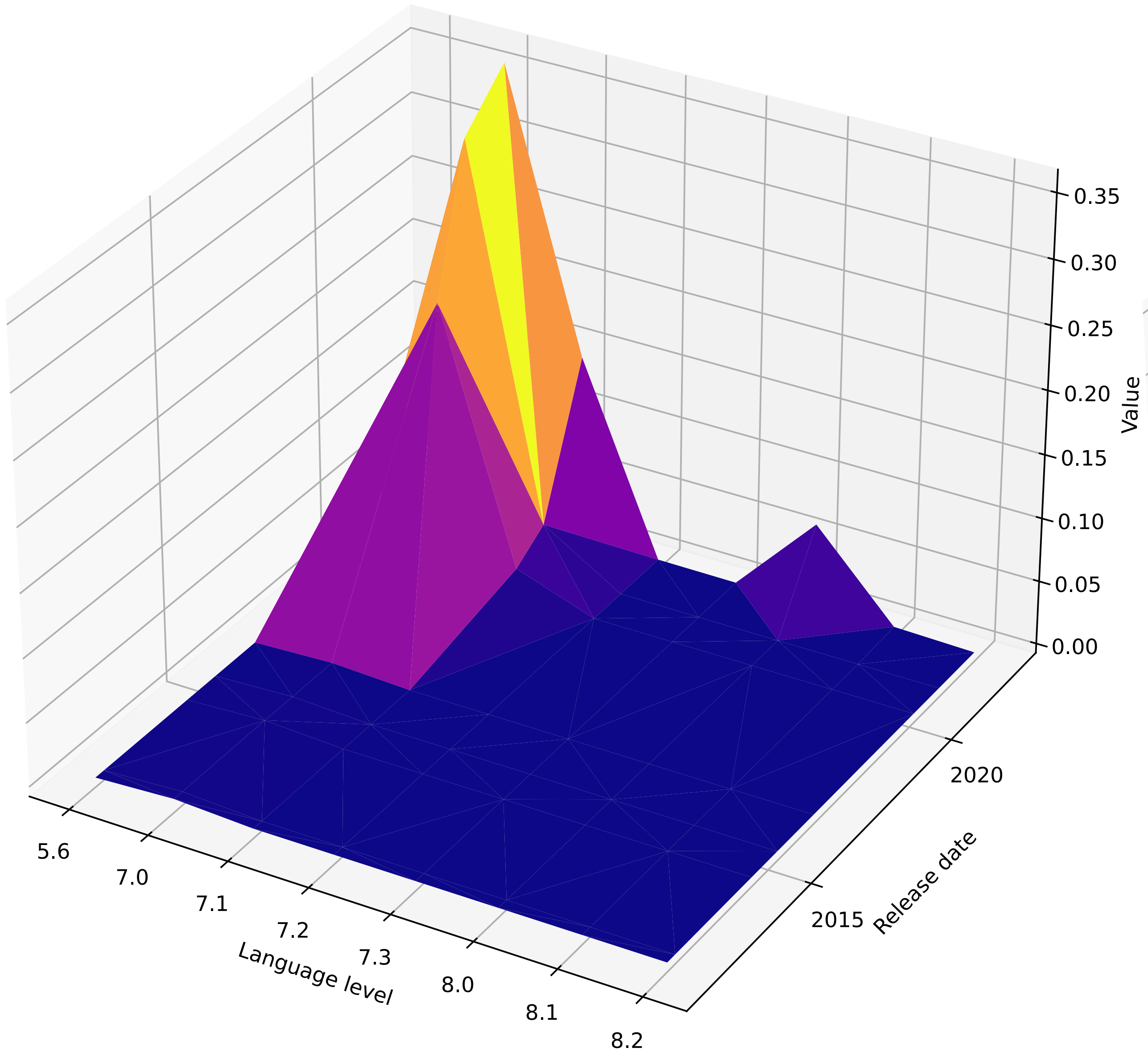
cf. https://doi.org/10.1109/SANER.2016.92

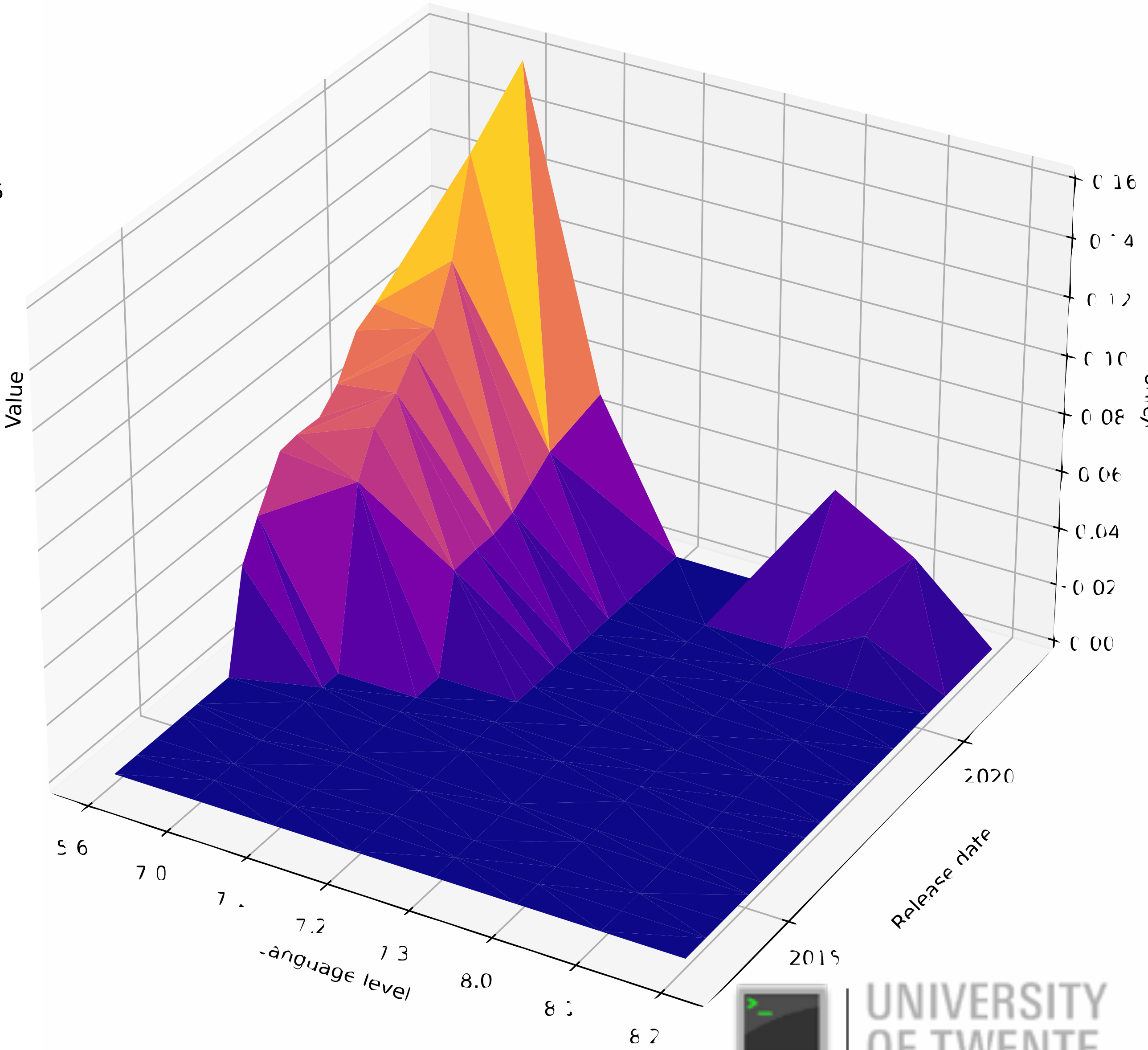
Portfolio/Codebase History



● IEF (1990-1996) ● Composer (1996-1997) ● COOL:GEN (1997-2004)
● Advantage Gen (2004-2012) ● CA Gen (2012-2015) ● Other

Evolution/Mining Perspective





That's What We Did!

- parse the source code
- obtain trees
- count modernity per node
  - based on children
- weighted sum per node type
- normalise at every step

Generalisable to WAG

- imagine attribute grammars
  - with static attributes
    - to enable per-type computations
  - with weights
    - to auto-count alternatives
- many other applications
  - test generation, PCG in games, conversational AI, …
- to be continued!

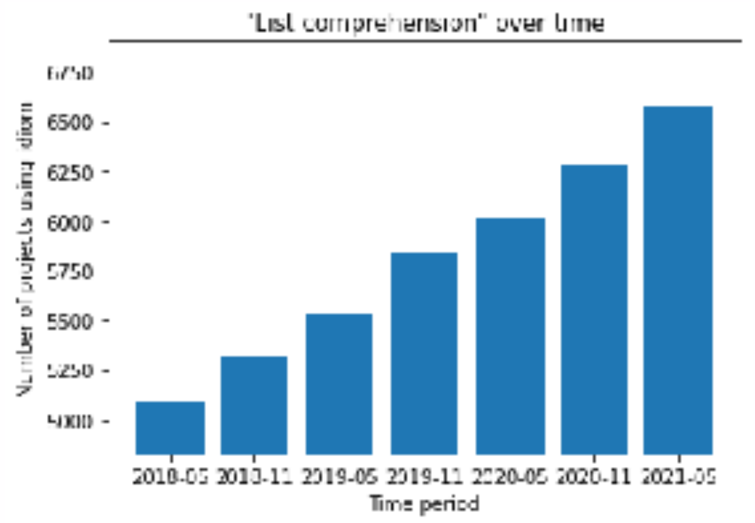
- language identification ⇒ modernity analysis
- weighted attribute grammars
- fingerprinting of codebases
- age ⇝ ench/maintenance, idioms/style, vulnerabilities
- info management / quantitative IT portfolio mgmt

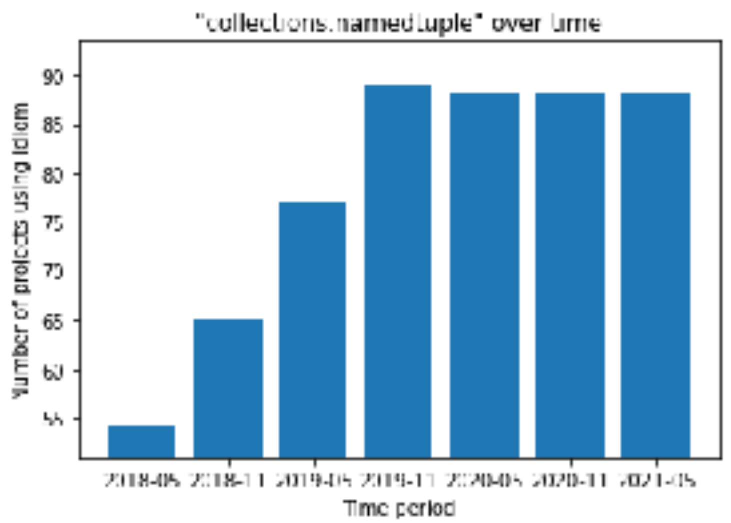
https://github.com/WoutervdBrink/PHP-Modernity-Signature

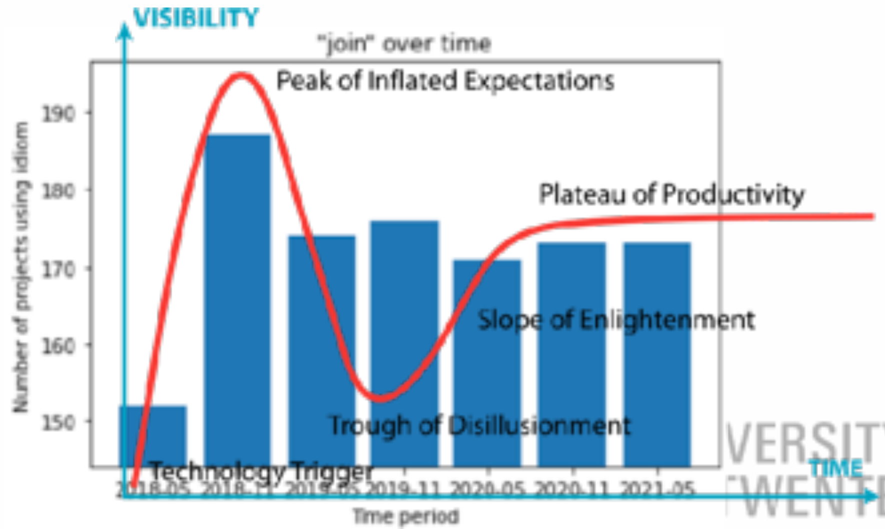
UNIVERSITY OF TWENTE.

Controversial

- Everything is better with grammars!
- Fingerprinting/profiling/…
- Other patterns of language feature adoption


"List comprehension" over time
Number of projects using idiom
Time period


"collections.namedtuple" over time
Number of projects using idiom
Time period


VISIBILITY
"join" over time
Peak of Inflated Expectations
Plateau of Productivity
Slope of Enlightenment
Trough of Disillusionment
Technology Trigger
TIME
Number of projects using idiom
Time period